High Performance SqueezeNext for CIFAR-10

Jayan Kant Duggal and Mohamed El-Sharkawy
Electrical and Computer Engineering
IoT Collaboratory, Purdue School of Engineering and Technology, IUPUI
jaydugga@iu.edu, melshark@iupui.edu

***Abstract*—CNNs is the foundation for deep learning and computer vision domain enabling applications such as autonomous driving, face recognition, automatic radiology image reading, etc. But, CNN is a algorithm which is memory and computationally intensive. DSE of neural networks and compression techniques have made convolution neural networks memory and computationally efficient. It improved the CNN architectures and made it more suitable to implement on real-time embedded systems. This paper proposes an efficient and a compact CNN to ameliorate the performance of existing CNN architectures. The intuition behind this proposed architecture is to supplant convolution layers with a more sophisticated block module and to develop a compact architecture with a competitive accuracy. Further, explores the bottleneck module and squeezenext basic block structure. The state-of-the-art squeezenext baseline architecture is used as a foundation to recreate and propose a high performance squeezenext architecture. The proposed architecture is further trained on the CIFAR-10 dataset from scratch. All the training and testing results are visualized with live loss and accuracy graphs. Focus of this paper is to make an adaptable and a flexible model for efficient CNN performance which can perform better with the minimum tradeoff between model accuracy, size, and speed. Finally, the conclusion is made that the performance of CNN can be improved by developing an architecture for a specific dataset. The purpose of this paper is to introduce and propose high performance squeezenext for CIFAR-10.**

***Index Terms*—Convolution Neural Networks (CNN), Deep Neural Networks (DNN), Design Space Exploration (DSE), High Performance SqueezeNext SqueezeNext, SqueezeNet, Pytorch, CIFAR-10.**

I. Introduction

In past few years, the performance of CNN has been improving at an exponential pace, majorly due to new methodologies, techniques, algorithms and improved network architectures, powerful hardware, and larger datasets. Due to resource high computational and memory complexity, leading to resource scarcity, there is a need of a small and compact CNN architectures. There are several advantages of small CNN architectures such as less model update overhead, low model size, low time complexity, and better feasibility for hardware deployment [1]. Therefore, the optimization of DNNs is a need of the hour. Hence, a great focus is laid, directly on the improving the CNNs with a few parameters, in order to be deployable on real time embedded platforms with a good model performance. SqueezeNet and SqueezeNext baseline architectures are those macro CNN architectures which fulfill the requirement of compact, small and efficient CNNs. A great emphasis is laid on the deployability of these models on real-time hardware systems such as advanced driver-assistance systems (ADAS), drones, edge devices, robotics, UAVs, and all other real-time applications that require low-cost and low-power. In Section 2 of this paper, the related architectures, SqueezeNet and SqueezeNext architectures along with their observed problems were reviewed. Subsequently, in Sections 3, the general methods to improve CNN performance such as architecture tuning, different learning rate methods, save and load checkpoint method, different types of optimizers and different activation functions were discussed. This section reflects the DSE of DNNs which is essential in building or improving any CNN/DNN. Section 4, defines the hardware and software used. In section 5, proposed High Performance SqueezeNext architecture is discussed along with a brief comparison with other CNN modules. Further, section 6 presents the obtained results for all the architectures mentioned in this paper were discussed in accordance with the model accuracy, size and speed. The attention in this paper is more focused to understand how the small and compact CNN architecture's design choices impact model accuracy, size, and speed. Hence, the insights gained from the design space exploration of the squeezenet and squeezenext baseline architectures (section 2 section 3) were used for the creation or development of the proposed high performance squeezenext architecture. Finally, the paper conclusion is made in section 7 that mentions the wholesome overview of the paper.

II. Related Architectures

A. SqueezeNet Architecture

This section reviews the squeezenet architecture [2] which comprises of fire modules, Relu activation, max and average pool layers, softmax activation, and kaiming uniform initialization. Fire module is the backbone of this architecture, comprising of a squeeze layer, s2 (1x1) and two expand layers, e1 (1x1) and e3 (3x3). The three following design strategies are employed to construct the baseline squeezenet architecture :

Strategy1 Replace 3x3 filters with 1x1 filters.

Strategy2 Decrease the number of input channels to 3x3 filters.

Strategy3 Down sample late in the network.

Fire modules greatly reduce the number of parameters as compared to the state of the art VGG architectures. The VGG architecture with 385MB model size was reduced

This is the author's manuscript of the article published in final edited form as:

Duggal, J. K., & El-Sharkawy, M. (2019). High Performance SqueezeNext for CIFAR-10. 2019 IEEE National Aerospace and Electronics Conference (NAECON), 285–290. https://doi.org/10.1109/NAECON46414.2019.9058217

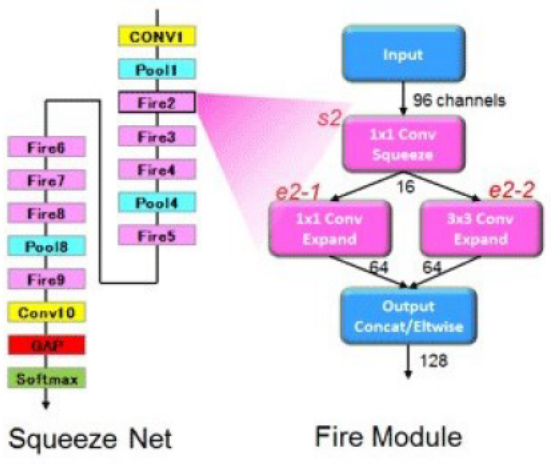

Fig. 1. Left: Squeezenet architecture, Right: Fire module.

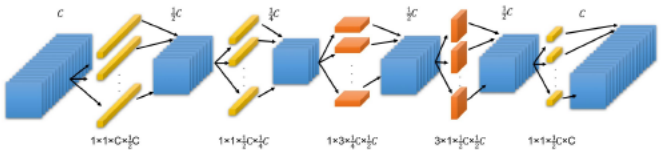

Fig. 2. SqueezeNext two stage bottleneck module illustration [2].

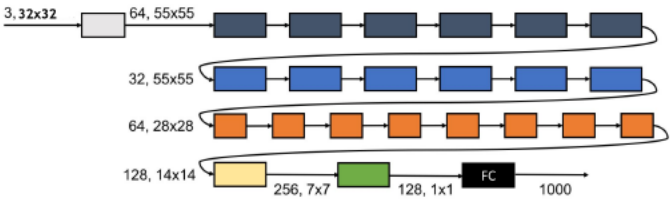

Fig. 3. Squeezenet baseline architecture with stage implementation of [6,6,8,1] configuration [2].

down to 2.5 MB of squeezenet baseline's model size with decent accuracy. The squeezenet architecture has a scope of further improvement with the help of using the following methods such as batch normalization layers, element-wise addition skip connections, in-place operations and other types of the optimizers. Figure 1 illustrates the squeezenet baseline architecture along with the representation of fire module.

B. SqueezeNext Architecture

Squeezenext baseline architecture [3] emerged after the advent of the squeezenet architecture. Squeezenext uses squeezenet baseline as its foundation and consists of the following strategies:

(1) More aggressive channel reduction by introducing a two-stage squeeze module, further reducing number of parameters used with the 3x3;

(2) 3x3 separable convolutions, and remove the additional 1x1 branch after the squeeze module to reduce the model size;

(3) An element-wise addition skip connection similar to ResNet architecture [4].

Squeezenext baseline architecture comprises of bottleneck modules with four stage implementation, batch normalization layers, Relu and Relu (in-place) nonlinear activations, max, and average pool layers, Xavier uniform initialization, a spatial resolution layer and a fully connected layer in the last with this [6,6,8,1] four stage block configuration. Bottleneck module shown in Figure 2 is the backbone of the squeezenext architecture as it significantly reduces the number of parameters without the deterioration of the model accuracy. In fact, a better model accuracy and size is attained in comparison to squeezenet baseline architecture. The squeezenext baseline [6,6,8,1] architecture configuration shown in Figure 3 that illustrates the squeezenext baseline architecture implemented on the CIFAR-10 dataset with input size 32x32 and 3 input channels.

This is the input for the first convolution, the white block. Now, the output of the first convolution is the input for a max pooling layer after the first convolution, not shown in Figure 3, but shown in Figure 6. The consecutive different colored blocks that are dark blue, blue, orange and yellow blocks after the first convolution and max-pooling represents the four-stage configuration implementation followed by a green block, representing the spatial resolution layer and the average pooling layer. Finally, followed by one black block that is a fully connected layer. The color change in the four stage implementation configuration blocks that are dark blue, blue, orange and yellow blocks depict a change in the input feature map's resolution. The squeezenext basic block structures shown in Figure 6 are the building blocks of the squeezenext baseline architecture. The basic block structure depicts each of the first blocks, that means the first dark

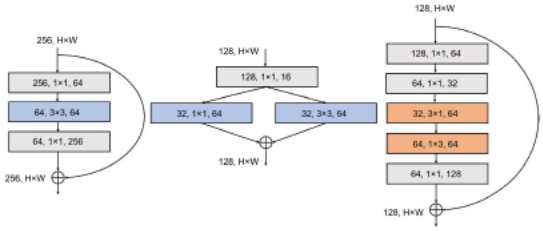

Fig. 4. Illustration of architecture baseline modules of ResNet, SqueezeNet, SqueezeNext [2].

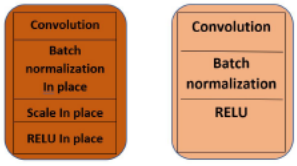

Fig. 5. Left: Squeezenext baseline basic-block module, Right: Squeezenext pytorch basic-block module iuuuyi's version [10].

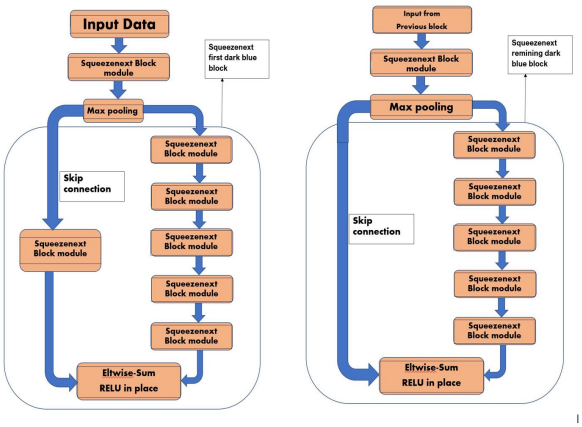

Fig. 6. Left: Squeezenext first block structure with a squeezenext block module within skip connection, Right: Squeezenext second block structure with skip connection used after first block structure in all stages of four-stage implementation. All squeezenext block modules here refer to baseline block module.

blue, blue, orange and the last yellow block of the four-stage implementation configuration is shown on the left side of Figure 6 and the basic block structure for each of the remaining blocks of the four stage implementation configuration is shown on the right side of Figure 6.

III. Methods to improve CNN performance

Before addressing the methods of improvement to be made in the baseline architecture of squeezenext to propose the high performance squeezenext architecture, the awareness of the general methods for the performance improvement of a convolution neural network is established before hand. The performance can be improved in the following ways.

1) Improve the performance with data . This refers to collect and/or invent more data, improve the quality of the data, data augmentation, and feature selection techniques.

2) Improve the performance with the architecture tuning which can be done by model diagnostics, tuning or tweaking with the following techniques such as weight initialization, learning rate, activation functions, network topology, regularization, different optimization and loss techniques.

3) Improve the performance with the architecture modification. In this method, new architecture can be inspired by the literature review, benefits of the existing architectures and re-sampling techniques.

4) Improve the performance with ensembles which include the following possible ways that are to combine models, combine views, and stacking.

Only the second and third methods are within the scope of this paper, exclusively.

A. Architecture tuning

The following ideas used in this paper for tuning the architecture are :

1) Different Learning Rate Schedule.

2) Save and Load checkpoint.

2) Use of different optimizers.

3) Different activation functions.

B. Different Learning Rate Schedule methods

Learning rate schedules seek to adjust the learning rate during training by reducing the learning rate according to a pre-defined schedule. Common learning rate schedules include time-based decay, step decay, exponential decay, and cosine annealing. Figure 7 illustrates step decay based learning rate performs better than other learning rate schedule methods [8].

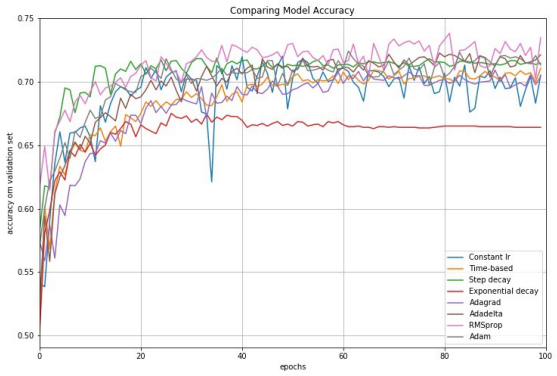

Fig. 7. Comparison of different LR Scheduling methods and optimizers source: [https://towardsdatascience.com/learning-rate-schedules-and-adaptive-learning-rate-methods-for-deep-learning-2c8f433990d1].

C. Save and Load checkpoint method

For improving accuracy, save and load the model method was used. Optimizer state dictionary and sometimes, other items such as epochs, loss, accuracy, net module, embedding layers, etc are also saved and loaded. The optimizer state dictionary contains additional updated information of buffers and parameters. Layers with only the learnable parameters have entries in the model state dictionary while the optimizer state dictionary contains information about the optimizers state and all the hyperparameters used. For the efficient model size and speed, the models state dictionary is only saved and loaded. In pytorch, torch.save(), torch.load() functions are used for saving and loading the state dictionaries in a checkpoint file.

D. Use of different optimizers

In this paper, different optimizers were implemented on proposed high performance squeezenext architecture based on the insights from the paper [7]. The following optimizers are generally used such as SGD (Stochastic Gradient Descent), ASGD (Averaged Stochastic Gradient Descent), Adam (Adaptive Moment Estimation), Adagrad (Adaptive subgradient methods for Online Learning and Stochastic Optimization), RMSprop (RMSprop algorithm), Rprop (Resilient back propagation algorithm). Refer [7] for the

mathematical form or equations of the optimizers.

SGD performs a parameter update, one update at a time, for each training example. The problem with SGD is the frequent updates with high variance, it causes the objective function to fluctuate heavily, ultimately complicates the convergence to the exact minimum and has trouble navigating ravines. Momentum is introduced to deal with these problems, to help accelerate SGD in the relevant direction and dampens oscillations. As a result, faster convergence is gained.

Adagrad makes big and small updates for infrequent and frequent parameters, respectively. So, it is well-suited for dealing with sparse data. It uses a different learning rate(LR) for every parameter based on the past gradients, hence, manually tuned learning rate was not required. But the weakness of Adagrad is that the LR is always decreasing and decaying, due to the accumulation of each squared gradients in the denominator. It causes the learning rate to shrink and eventually, the model stops learning entirely, gives very slow convergence, long to train and learn.

Adadelta seeks to reduce its aggressive, monotonically decreasing learning rate. Instead of accumulating all the past squared gradients, it restricts the window of accumulated past gradients to some fixed size. The sum of gradients is recursively defined as a decaying average of all past squared gradients. The running average depends only on the previous average and the current gradient. The benefit of using this is that there is no need to set a default learning rate.

RMSprop divides the learning rate by an exponentially decaying average of squared gradients. It suggests the momentum and the learning rate to be set at the default the values of 0.9 and 0.001, respectively.

Adam store both factors of exponentially decaying average of past squared gradients and exponentially decaying average of past gradients and computes adaptive learning rates for each parameter. The learning speed of the model is fast and efficient. It rectifies problems of optimization techniques such as vanishing learning rate, slow convergence or high variance. The problem observed is that, initially, the algorithm performs poorly on the discussed related architectures.

Adamax is a special case of Adam where its second order moment is replaced by infinite order moment. Infinite order norm makes the algorithm more stable according to this algorithm. The bias is not computed, hence, it is not suggested to bias towards zero. Good default values for learning rate, beta 1 and beta 2 are 0.002, 0.9, and 0.999, respectively.

Rprop is similar to back-propagation with the advantages such as fast training, doesn't require to specify any free parameter values, adapts the step size dynamically for each weight, independently. The disadvantage of Rprop is that it is a more complex algorithm to implement. It generally requires large batch updates and the step sizes jump around too much and updates work badly if there is too much randomness.

E. Different activation functions

Rectified Linear Units (ReLU) [9] is not a linear and provides the same benefits as sigmoid activation but with better performance. The mathematical formula is max (0,z). It avoids and rectifies the vanishing gradient problem, less computationally expensive and involves simpler mathematical operations. But, it can blow up the activation due to its range [0, inf). Also, One of its limitations is that it should only be used within Hidden layers of a Neural Network Model. **Exponential Linear Unit** (ELU) [9] is a function that tends to converge cost to zero faster and produce more accurate results. ELU has an extra alpha constant which should be a positive number. It is very similar to RELU except for negative inputs. They are both in the identity function form for non-negative inputs. ELU becomes smooth slowly until its output equal to - whereas RELU sharply smooths. But, for it can blow up the activation with the output range of [0, inf] for x greater than zero.

IV. Hardware and Software used

- Intel i7 8th generation processor with 32 GB RAM.
- Required memory for dataset and results: 4GB.
- NVIDIA GTX 1080 GPU.
- Spyder version 3.6.
- Pytorch version 1.0.
- Livelossplot (Loss and accuracy visualization).
- Architecture visualization : Netscope.

V. Proposed High Performance Squeezenext Architecture

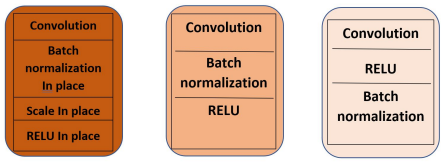

Fig. 8. Extreme right illustrates the proposed high performance squeezenext basic block module.

The proposed high performance squeezenext architecture is inspired from the baseline squeezenext architecture [3] and the inspiration for implementation of the basic block module and ELU implementation within the proposed architecture is taken from the other two other papers [5], [6]. This architecture uses a different structure of basic block module shown in Figure 8 than the baseline squeezenext and squeezenext pytorch's implementation of basic block structure. The proposed high performance architecture basic block is compatible with pytorch. The bottleneck module is chosen over fire module for proposed high performance squeezenext architecture because as observed in Figure 4, the bottleneck module has a better parameter reduction then squeezenet and resnet module due to the fact that it uses a two stage bottleneck module to reduce the number of input channels down to 3x3 convolution. Further, in squeezenext

3X3 convolution in comparison to other modules is decomposed into 3x1 and 1x3 convolutions (orange blocks) which in turn, again reduces the number of parameters, followed by a 1x1 expansion module. The high performance squeezenext architecture comprises of bottleneck modules, activation layer, batch normalization layers, pooling layers (ceiling function is used instead of floor function) and a fully connected layer in the last with [1,1,1,1] four stage implementation configuration with 0.5x network width multiplier are used for better model performance. The proposed architecture basic building block structures for the CIFAR-10 dataset is shown in Figure 9, and these structures are used in similar manner as used in the baseline architecture configuration as shown in Figure 3 but the number of blocks in the four stage implementation configuration and network width multiplier are changed depending on the architecture end application. The basic block module in Figure 8 and the basic building block structure modules shown in Figure 6, together implemented in [1,1,1,1] four stage implementation configuration with network width multiplier 0.5x forms the complete proposed high performance squeezenext architecture.

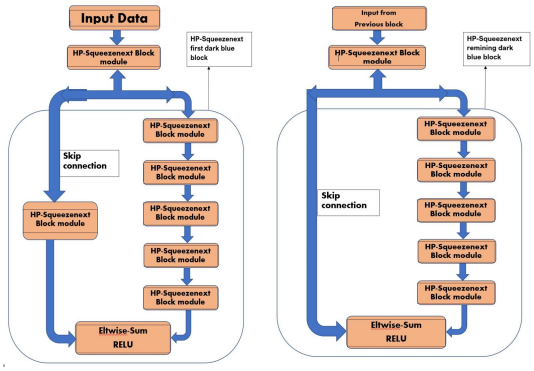

Fig. 9. Basic building block structure modules for proposed high performance architecture.

VI. Results

The obtained results are discussed in this section. The proposed architecture is trained and tested from scratch on the CIFAR-10 dataset to improve the overall performance of the proposed architecture. All the results obtained in this paper were implemented with the following common hyperparameter values: 0.1, batch size: 128, weight decay: 5e-4, total number of epochs: 200, standard cross entropy loss function and with a live accuracy and loss graphs update livelossplot package.

A. Model accuracy improvement

Other existing algorithms and methodologies have attained better accuracy than the squeezenext architecture's modified version that is 92.09% accuracy, shown in Table 1. However, all those machine learning algorithms use transfer learning techniques, in which the model is first trained on a large dataset, ImageNet and then pre-trained model is fine-tuned on a smaller dataset like CIFAR-10. Additionally these architectures also use data augmentation techniques. The transfer learning technique provides better accuracy than a network trained from scratch due to that reason. However, training the network using the CIFAR-10 dataset takes less time and computation power. Therefore, the models mentioned in this paper, are the networks which were trained from scratch on the CIFAR-10 dataset and without use of any data augmentation. To improve the accuracy of the squeezenext pytorch architecture, the save and load checkpoint method is implemented along with the architecture modification with a kernel size 3x3 and stride 1. Also, the specific step time learning rate schedule with the exponential update is also used. Table 1 and Figure 10 compared the results obtained for the modified architecture, the proposed high performance squeezenext architecture with the baseline squeezenet and squeezenext architectures.

B. Model size and speed improvement

The model speed in this paper means the per epoch time cost of training and testing the architecture on CIFAR-10 dataset. In general, more powerful hardware (better GPU or multiple GPUs), architecture pruning, and methods discussed in Section 3 are used to improve a CNN model size and speed. The CIFAR-10 dataset is quite small as compared to Imagenet so the model depth, as well as the width, is reduced for better model performance. The proposed high performance squeezenext is implemented with low network depth and width multipliers, in-place activation layers, element-wise operations, no max-pooling layers were used only average pooling is used in the last just before the FCC layer. All the HP-SqueezeNext architectures uses [1,1,1,1] four stage implementation configuration with a different network width along with following hyperparameters such as SGD optimizer with momentum and nestrov values equal to 0.9 and true, Step LR decay schedule comprising of four different learning rates with an exponential LR update were used to train and test the CIFAR-10 dataset. Also, optimizer and other additional state dictionary were not saved in the model checkpoint file, only net state dictionary is saved. Minimum achieved model size is 370KB with a model speed of 7 seconds which is the average time cost for one epoch. The proposed architecture trained and tested the whole CIFAR-10 dataset in 28 minutes, then to the squeezenext baseline architecture and squeezenext pytorch implementation which takes up to 1 hours 21 minutes and 2 hours 42 minutes, respectively. The proposed high performance architecture is able to achieve both better model size and model speed with decent accuracy. Though, a better model size and speed of 117 KB and 5 seconds respectively, is also achieved with a model width of 0.125x for the proposed high performance squeezenext architecture, but the model accuracy reduced to 61.87%.

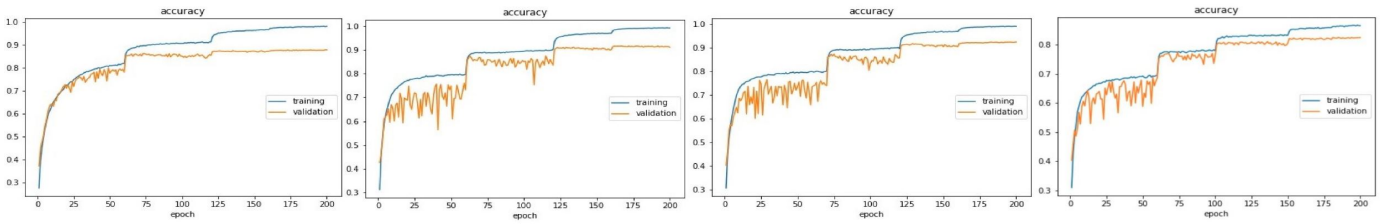

Fig. 10. 1: Squeezenext baseline accuracy, 2: Squeezenext iuuuyi's pytorch arch accuracy, 3: Squeezenext pytorch architecture modified accuracy (best accuracy result), 4: High Performance Squeezenext-06-1x-v1 accuracy (best model size and speed).

TABLE I
MODEL PERFORMANCE IMPROVEMENT

Model	Accuracy%	Model size(MB)	Model speed(sec)
SqueezeNet-v1-0(baseline)	78.1	2.75	7
HP-SqueezeNext-06-0.50x-v1	82.44	0.370	7
SqueezeNext-23-1x-v1(baseline)	87.56	2.59	23
HP-SqueezeNext-06-1x-v1	86.82	1.24	8
SqueezeNet-23-1x-v1(iuuuyi)	91.68	2.58	48
SqueezeNext-23-1x-v1(modified)	92.25	5.14	48
HP-SqueezeNext-06-0.75x-v1	82.86	1.24	8
HP-SqueezeNext-21-1x-v2	92.05	2.60	18
HP-SqueezeNext-19-1x-v1	88.54	2.80	18

*All results are 3 average runs with SGD, LR is 0.1

VII. CONCLUSION

The purpose of this research paper is to introduce the proposed high performance squeezenext architecture. Squeezenext baseline is the underlying foundation of the proposed architecture. Best model accuracy of 92.05% is obtained that is 14% better than the SqueezeNet baseline and 4.5% better than SqueezeNext baseline. A model size (time cost for training and testing per epoch on GPU) of 0.370 KB is achieved which is 7.5x better than SqueezeNet baseline, 7x better than SqueezeNext baseline and almost, 14x better than proposed implementation of SqueezeNext for CIFAR-10. Further, a model speed of 7 seconds that is 16 seconds better than SqueezeNext baseline, 41 seconds better than SqueezeNext proposed modified implementation and equivalent to SqueezeNet baseline model speed. Although, a better model accuracy, 92.25% (0.20% better than the proposed High Performance Squeezenext), was achieved for proposed modified implementation of SqueezeNext baseline for CIFAR-10 but we can observe that this model has big model size which is more than 5MB (crosses the bench of SqueezeNet and SqueezeNext baseline) and also, model speed takes a lot of time to train and test on GPU, too. This paper concludes that it is important to load the saved CNN model with the help of a model state dictionary method rather than saving all other hyper parameters, ultimately making use of optimizer state dictionary. For better accuracy, it is not necessary to save all the model parameters for an efficient model size and model speed. The choice of optimizers, initialization for convolution, batch normalization layers, and learning rate decay schedule affects the model's performance. The use of in-place functions improved model performance i.e. model size and model speed. Further, it also can be improved by training and testing the model for a specific dataset from scratch without data augmentation. All of the above mentioned qualities make the proposed architecture a flexible architecture considering the tradeoff between the model size, speed and accuracy.

VIII. ACKNOWLEDGEMENT

The authors would like to show their gratitude to Yi Lu for sharing his pytorch re-implementation of squeezenext baseline architecture for the research.

REFERENCES

[1] Szegedy, Christian, et al. "Inception-v4, inception-resnet and the impact of residual connections on learning." Thirty-First AAAI Conference on Artificial Intelligence. 2017.
[2] Iandola, Forrest N., et al. "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and¡ 0.5 MB model size." arXiv preprint arXiv:1602.07360 (2016).
[3] Gholami, Amir, et al. "Squeezenext: Hardware-aware neural network design." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops. 2018.
[4] He, Kaiming, et al. "Deep residual learning for image recognition." Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.
[5] Clevert, Djork-Arn, Thomas Unterthiner, and Sepp Hochreiter. "Fast and accurate deep network learning by exponential linear units (elus)." arXiv preprint arXiv:1511.07289 (2015).
[6] Shah, Anish, et al. "Deep residual networks with exponential linear unit." arXiv preprint arXiv:1604.04112 (2016).
[7] Ruder, Sebastian. "An overview of gradient descent optimization algorithms." arXiv preprint arXiv:1609.04747 (2016).
[8] Bengio, Yoshua. "Practical recommendations for gradient-based training of deep architectures." Neural networks: Tricks of the trade. Springer, Berlin, Heidelberg, 2012. 437-478.
[9] Pedamonti, Dabal. "Comparison of non-linear activation functions for deep neural networks on MNIST classification task." arXiv preprint arXiv:1804.02763 (2018).
[10] Deng, Jia, et al. "Imagenet: A large-scale hierarchical image database." (2009): 248-255.
[11] luuuyi, Yi Lu. Squeezenext pytorch re-implement version, (2018). GitHub repository, https://github.com/luuuyi/SqueezeNext.PyTorch
[12] Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556 (2014).